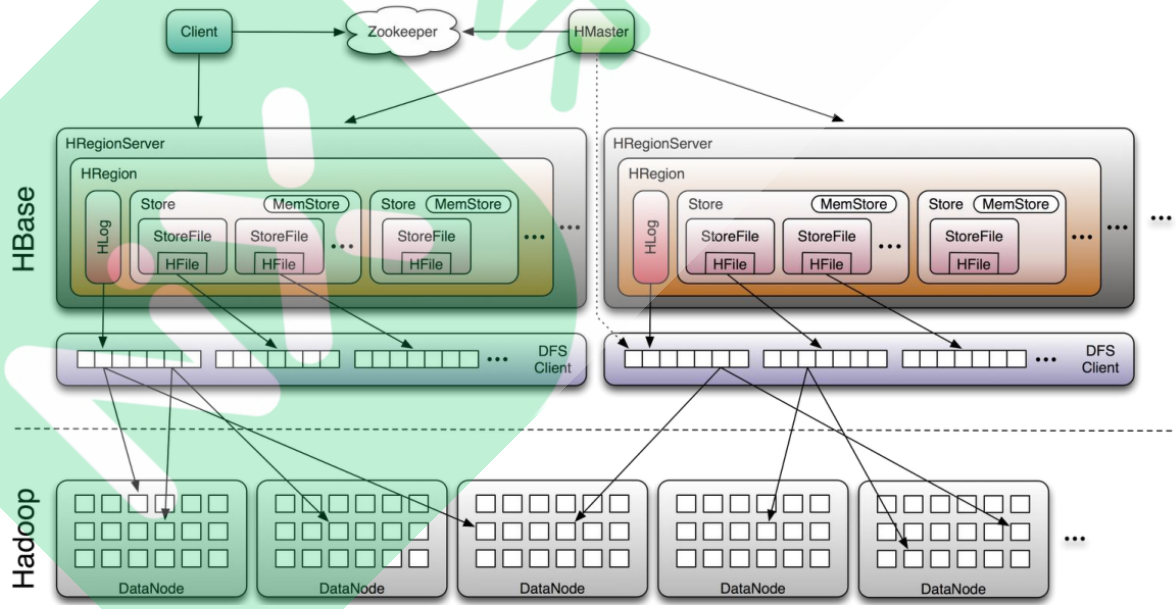
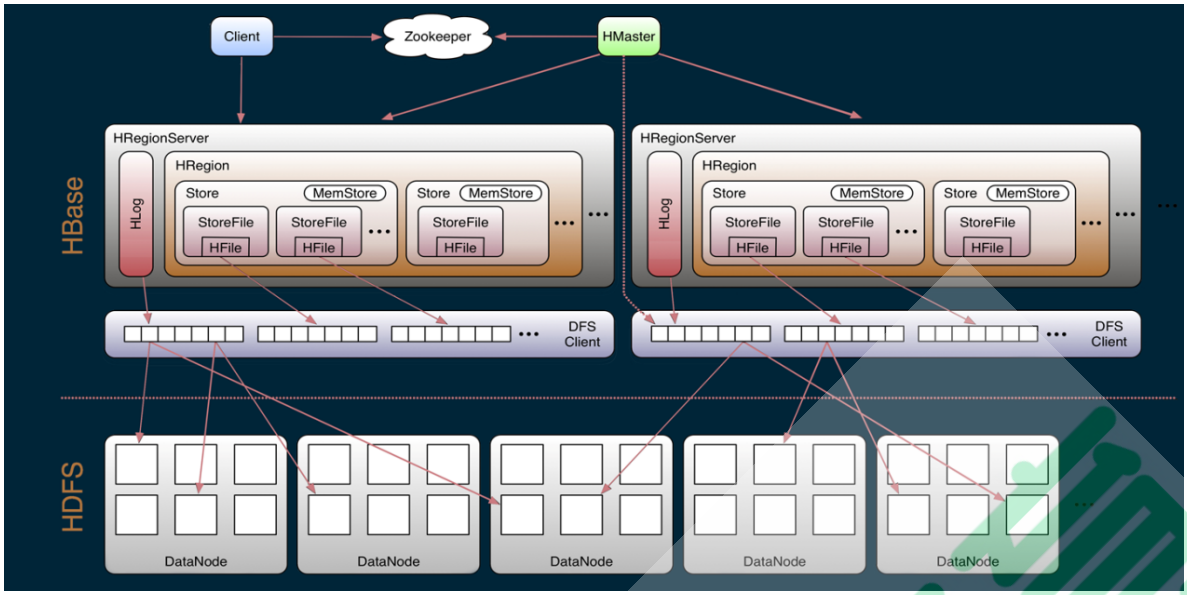
1. HBase底层原理

1.1. 系统架构





Client职责

1、HBase有两张特殊表：
.META.：记录了用户所有表拆分出来的的Region映射信息，.META.可以有多个Regoin
-ROOT-：记录了.META.表的Region信息，-ROOT-只有一个Region，无论如何不会分裂

2、Client访问用户数据前需要首先访问ZooKeeper，找到-ROOT-表的Region所在的服务器位置，然后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的服务器位置去访问，中间需要多次网络操作，不过client端会做cache缓存。

ZooKeeper职责

1、ZooKeeper为HBase提供Failover机制，选举Master，避免单点Master单点故障问题
2、存储所有Region的寻址入口：-ROOT-表在哪台服务器上。-ROOT-这张表的位置信息
3、实时监控RegionServer的状态，将RegionServer的上线和下线信息实时通知给Master
4、存储HBase的Schema，包括有哪些Table，每个Table有哪些Column Family

HBase的元数据有两种：

1、hbase中的表的相关信息，主要是指 表名，列簇的定义 等等， master 管理的
所有的关于创建表，删除表，修改表，等操作，必须经过 master 来实现
2、所有的用户表的regoin的位置信息，这种数据也称之为元数据
存储在meta表， meta表由多个region组成，这些region也会分散到各个regoinserver去存储

Master职责

1、为RegionServer分配Region
2、负责RegionServer的负载均衡
3、发现失效的RegionServer并重新分配其上的Region
    master是管理者，一个hbase系统会有很多表，每个表又有很多region，那么这些region到底交给那些region来管理就是由 master来决定
4、HDFS上的垃圾文件（HBase）回收
    region会compact也会split，必然会有失效的数据
5、处理Schema更新请求（表的创建，删除，修改，列簇的增加等等）
    这些关于schema的数据都是存储在ZOOKEeper，但是是master是负责更新的
    如果涉及到表的创建，修改，删除等操作，master宕机了就没法做，但是数据的插入和查询还是可以继续做

hbase集群中的master宕机一段时间，集群的服务会中断么？

RegionServer职责

1、RegionServer维护Master分配给它的Region，处理对这些Region的IO请求
2、负责和底层的文件系统HDFS的交互，存储数据到HDFS
    每个regionserver内部都有一个客户端（datnaode的代理）。 负责把数据写入到HDFS
3、负责Store中的HFile的合并Compact工作 + split工作
4、RegionServer负责Split在运行过程中变得过大的Region，负责Compact操作
    SplitPolicy 分割策略：有三个默认的策略！

总结:

可以看到，client访问HBase上数据的过程并不需要Master参与（寻址访问ZooKeeper和RegioneServer，数据读写访问RegioneServer），Master仅仅维护者Table和Region的元数据信息，负载很低。

.META. 存的是所有的 Region的位置信息，那么RegioneServer当中Region在进行分裂之后的新产生的Region，是由Master来决定发到哪个RegioneServer，这就意味着，只有Master知道new Region的位置信息，所以，由Master来管理.META.这个表当中的数据的CRUD

所以结合以上两点表明，在没有Region分裂的情况，Master宕机一段时间是可以忍受的。

如果master宕机了，那些事情不能做:

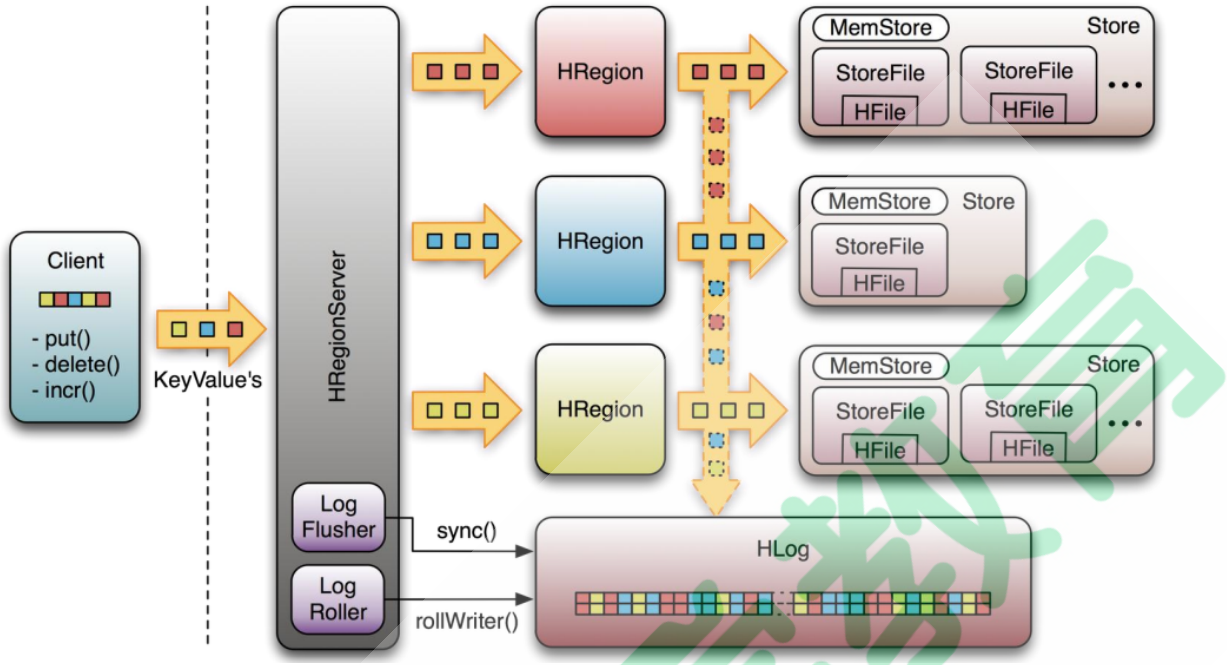
1、创建，修改，删除，表
2、负责均衡不能做
3、split不能做了。

那些操作能继续做呢?

1、读数据
2、写数据

1.2. 物理存储

1.2.1. 整体物理结构



WAL： write ahead log

要点知识：

1、Table中的所有行都按照RowKey的字典序排列。hbase的一个rowkey就对应一行数据

2、Table 在行的方向上分割为多个HRegion。

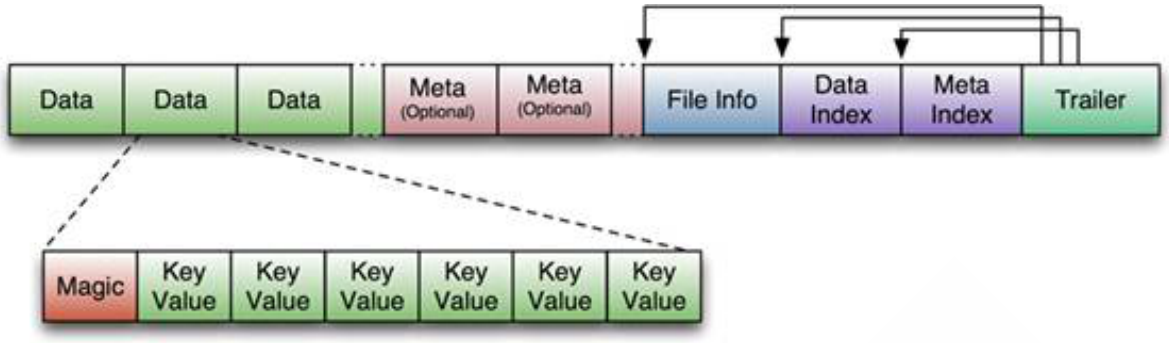
3、HRegion按大小分割的(默认10G)，每个表一开始只有一个HRegion，随着数据不断插入表，HRegion不断增大，当增大到一个阀值的时候，HRegion就会等分会两个新的HRegion。当表中的行不断增多，就会有越来越多的HRegion。

4、HRegion是HBase中分布式存储和负载均衡的最小单元。最小单元就表示不同的HRegion可以分布在不同的HRegionServer上。但一个HRegion是不会拆分到多个server上的。

5、HRegion虽然是负载均衡的最小单元，但并不是物理存储的最小单元。事实上，HRegion由一个或者多个Store组成，每个Store保存一个Column Family。每个Strore又由一个MemStore和0至多个StoreFile组成

1.2.2. StoreFile和HFile结构

StoreFile 以 HFile 格式保存在 HDFS 上，请看下图 HFile 的数据组织格式：



首先 HFile 文件是不定长的，长度固定的只有其中的两块：Trailer 和 FileInfo。
正如图中所示：
Trailer中有指针指向其他数据块的起始点。
FileInfo中记录了文件的一些Meta信息，例如：AVG_KEY_LEN, AVG_VALUE_LEN, LAST_KEY, COMPARATOR, MAX_SEQ_ID_KEY等。

HFile分为六个部分：

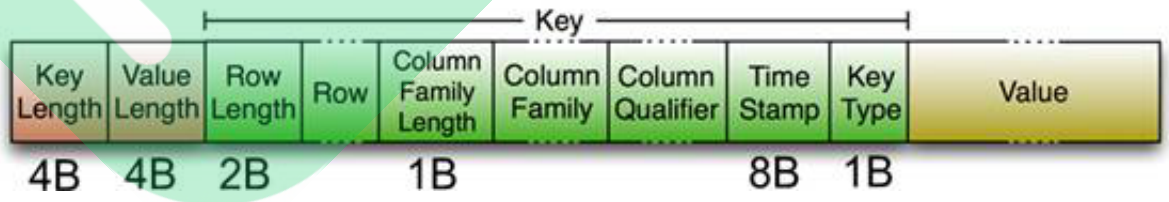
```
1、Data Block 段–保存表中的数据，这部分可以被压缩。
2、Meta Block 段 (可选的)–保存用户自定义的key-value对，可以被压缩。
3、File Info 段–HFile的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。
4、Data Block Index 段–Data Block的索引。每条索引的key是被索引的block的第一条记录的key。
5、Meta Block Index段 (可选的)–Meta Block的索引。
6、Trailer段–这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘IO将整个block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。
```

HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。
目标HFile的压缩支持两种方式：Gzip，LZO。

Data Index和Meta Index块记录了每个Data块和Meta块的起始点。

Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。 每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成，Magic内容就是一些随机数字，目的是防止数据损坏。

HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：




```
rowkey, cf, qualifier, timestamp, value
这四个字段的长度 + Cell的类型（Put/Delete）
```

开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示RowKey的长度，紧接着是 RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示TimeStamp和KeyType（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。

1.2.3. MemStore和StoreFile

一个HRegion由多个Store组成，每个Store包含一个列族的所有数据

Store包括位于内存的一个Memstore和位于硬盘的多个StoreFile组成

写操作先写入Memstore，当Memstore中的数据量达到某个阈值，HRegionServer启动flushcache进程写入Storefile，每次写入形成单独一个HFile

当总Storefile大小超过一定阈值后，会把当前的Region分割成两个，并由HMaster分配给相应的Region服务器，实现负载均衡

客户端检索数据时，先在Memstore找，找不到再找Storefile

1.2.4. HLog & WAL

WAL意为Write Ahead Log(http://en.wikipedia.org/wiki/Write-ahead_logging)，类似MySQL中的binlog，用来做灾难恢复之用，HLog记录数据的所有变更，一旦数据修改，就可以从Log中进行恢复。

HBase采用类LSM的架构体系，数据写入并没有直接写入数据文件，而是会先写入缓存（Memstore），在满足一定条件下缓存数据再会异步刷新到硬盘。为了防止数据写入缓存之后不会因为RegionServer进程发生异常导致数据丢失，在写入缓存之前会首先将数据顺序写入HLog中。如果不幸一旦发生RegionServer宕机或者其他异常，这种设计可以从HLog中进行日志回放进行数据补救，保证数据不丢失。HBase故障恢复的最大看点就在于如何通过HLog回放补救丢失数据。

WAL(Write-Ahead Logging)是一种高效的日志算法，几乎是所有非内存数据库提升写性能的不二法门，基本原理是在数据写入之前首先顺序写入日志，然后再写入缓存，等到缓存写满之后统一落盘。之所以能够提升写性能，是因为WAL将一次随机写转化为了一次顺序写加一次内存写。提升写性能的同时，WAL可以保证数据的可靠性，即在任何情况下数据不丢失。假如一次写入完成之后发生了宕机，即使所有缓存中的数据丢失，也可以通过恢复日志还原出丢失的数据。

每个Region Server维护一个HLog，而不是每个Region一个。这样不同region(来自不同table)的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对table的写性能。带来的麻烦是，如果一台region server下线，为了恢复其上的Region，需要将RegionServer上的log进行拆分，然后分发到其它RegionServer上进行恢复。

HLog文件就是一个普通的Hadoop Sequence File：

```
1、HLog Sequence File 的Key是HLogKey对象，HLogKey中记录了写入数据的归属信息，除了table和region名字外，同时还包括 sequence number和timestamp，timestamp是”写入时间”，sequence number的起始值为0，或者是最近一次存入文件系统中sequence number。

2、HLog Sequece File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue
```

WAL持久化等级

HBase 中可以通过设置 WAL 的持久化等级决定是否开启 WAL 机制、以及 HLog 的落盘方式。WAL 的持久化等级分为如下几个等级：

1、SKIP_WAL：只写缓存，不写HLog日志。这种方式因为只写内存，因此可以极大的提升写入性能，但是数据有丢失的风险。在实际应用过程中并不建议设置此等级，除非确认不要求数据的可靠性。

2、ASYNC_WAL：异步将数据写入HLog日志中。

3、SYNC_WAL：同步将数据写入日志文件中，需要注意的是数据只是被写入文件系统中，并没有真正落盘。

4、FSYNC_WAL：同步将数据写入日志文件并强制落盘。最严格的日志写入等级，可以保证数据不会丢失，但是性能相对比较差。

5、USER_DEFAULT：默认如果用户没有指定持久化等级，HBase使用SYNC_WAL等级持久化数据。

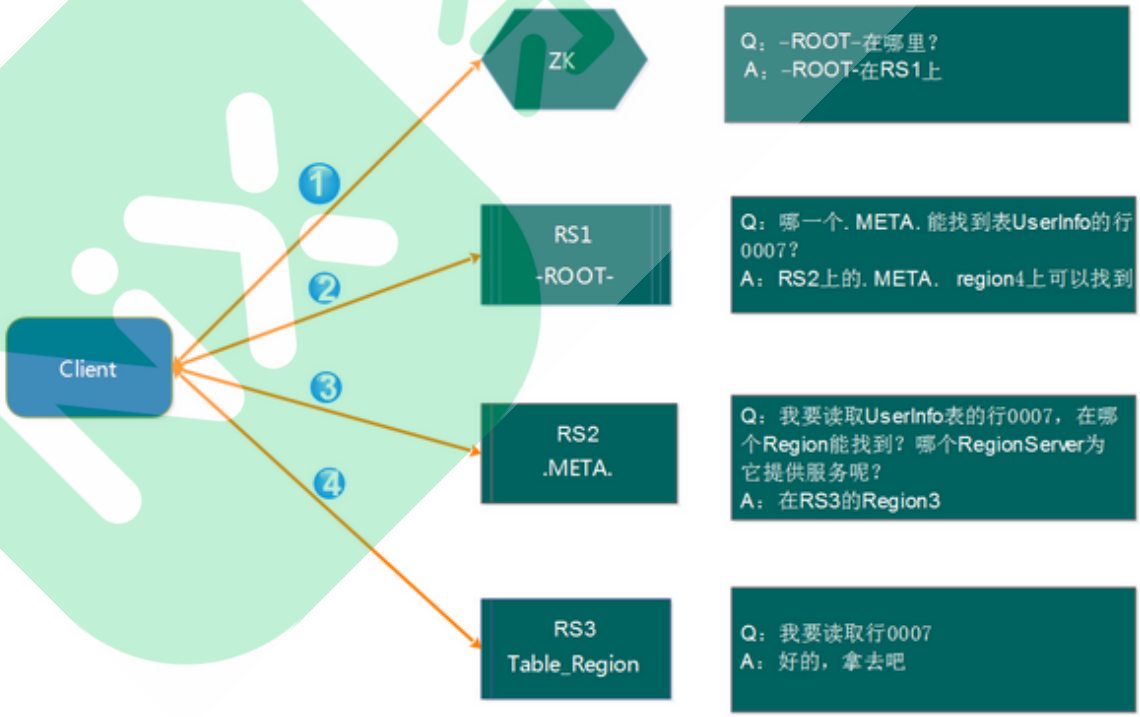
用户可以通过客户端设置WAL持久化等级，代码：put.setDurability(Durability. *SYNC_WAL*);

1.3. 寻址机制

既然读写都在RegionServer上发生，我们前面有讲到，每个RegionSever为一定数量的Region服务，那么Client要对某一行数据做读写的时候如何能知道具体要去访问哪个RegionServer呢？那就是接下来我们要讨论的问题

1.3.1. 老的Region寻址方式

在HBase-0.96版本以前，HBase有两个特殊的表，分别是-ROOT-表和.META.表，其中-ROOT-的位置存储在ZooKeeper中，-ROOT-本身存储了.META. Table的RegionInfo信息，并且-ROOT-不会分裂，只有一个Region。而.META.表可以被切分成多个Region。读取的流程如下图所示：



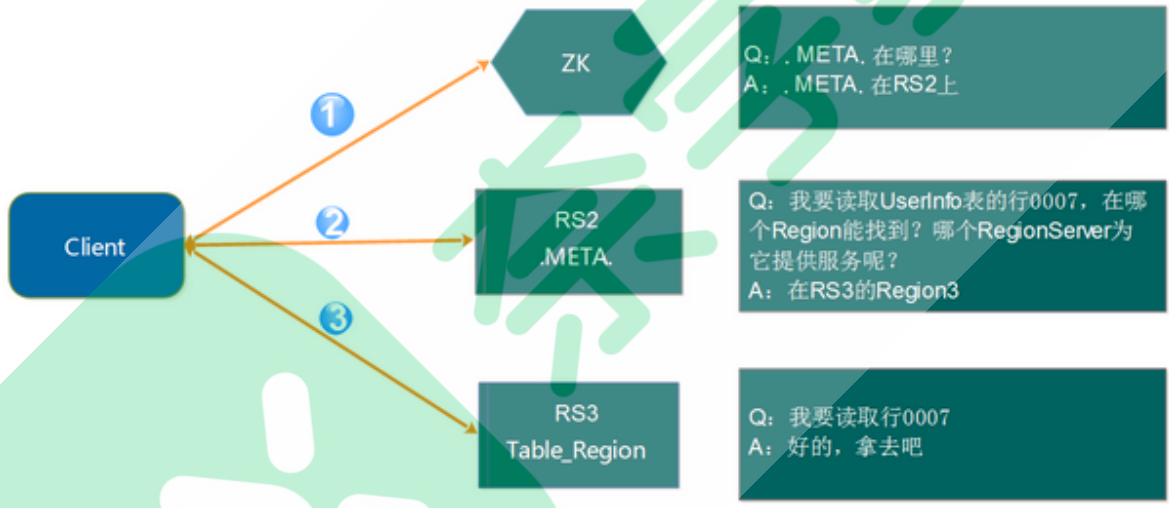
详细步骤：

```
第1步：Client请求ZooKeeper获得-ROOT-所在的RegionServer地址
第2步：Client请求-ROOT-所在的RS地址，获取.META.表的地址，Client会将-ROOT-的相关信息cache下来，以便下一次快速访问
第3步：Client请求.META.表的RegionServer地址，获取访问数据所在RegionServer的地址，Client会将.META.的相关信息cache下来，以便下一次快速访问
第4步：Client请求访问数据所在RegionServer的地址，获取对应的数据
```

从上面的路径我们可以看出，用户需要3次请求才能直到用户Table真正的位置，这在一定程序带来了性能的下降。在0.96之前使用3层设计的主要原因是考虑到元数据可能需要很大。但是真正集群运行，元数据的大小其实很容易计算出来。在BigTable的论文中，每行METADATA数据存储大小为1KB左右，如果按照一个Region为128M的计算，3层设计可以支持的Region个数为2^34个，采用2层设计可以支持2^17（131072）。那么2层设计的情况下一个集群可以存储4P的数据。这仅仅是一个Region只有128M的情况下。如果是10G呢? 因此，通过计算，其实2层设计就可以满足集群的需求。因此在0.96版本以后就去掉了-ROOT-表了。

1.3.2. 新的Region寻址方式

如上面的计算，2层结构其实完全能满足业务的需求，因此0.96版本以后将-ROOT-表去掉了。如下图所示：



访问路径变成了3步：

```
第1步：Client请求ZooKeeper获取.META.所在的RegionServer的地址。
第2步：Client请求.META.所在的RegionServer获取访问数据所在的RegionServer地址，Client会将.META.的相关信息cache下来，以便下一次快速访问。
第3步：Client请求数据所在的RegionServer，获取所需要的数据。
```

总结去掉-ROOT-的原因有如下2点：

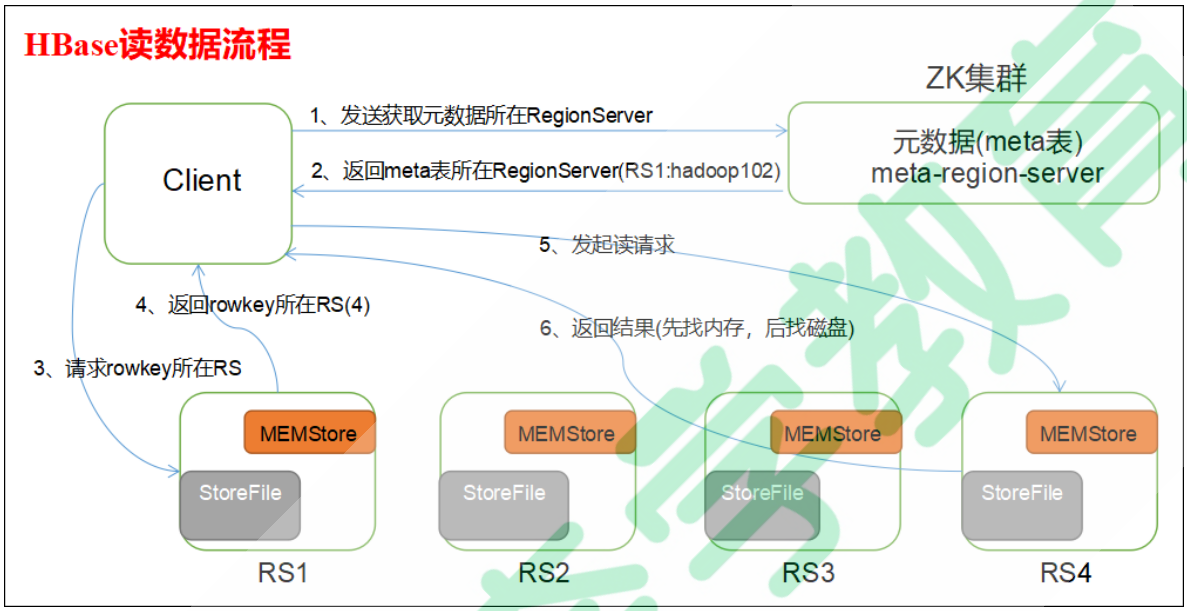
```
其一：提高性能
其二：2层结构已经足以满足集群的需求
```

这里还有一个问题需要说明，那就是Client会缓存.META.的数据，用来加快访问，既然有缓存，那它什么时候更新？如果.META.更新了，比如Region1不在RerverServer2上了，被转移到了RerverServer3上。Client的缓存没有更新会有什么情况？

其实，Client的元数据缓存不更新，当.META.的数据发生更新。如上面的例子，由于Region1的位置发生了变化，Client再次根据缓存去访问的时候，会出现错误，当出现异常达到重试次数后就会去.META.所在的RegionServer获取最新的数据，如果.META.所在的RegionServer也变了，Client就会去ZooKeeper上获取.META.所在的RegionServer的最新地址。

1.4. 读写过程

1.4.1. 读请求过程



详细步骤：

```
1、客户端通过ZooKeeper以及-ROOT-表和.META.表找到目标数据所在的RegionServer(就是数据所在的Region的主机地址)
2、联系RegionServer查询目标数据
3、RegionServer定位到目标数据所在的Region，发出查询请求
4、Region先在Memstore中查找，命中则返回
5、如果在Memstore中找不到，则在Storefile中扫描
```

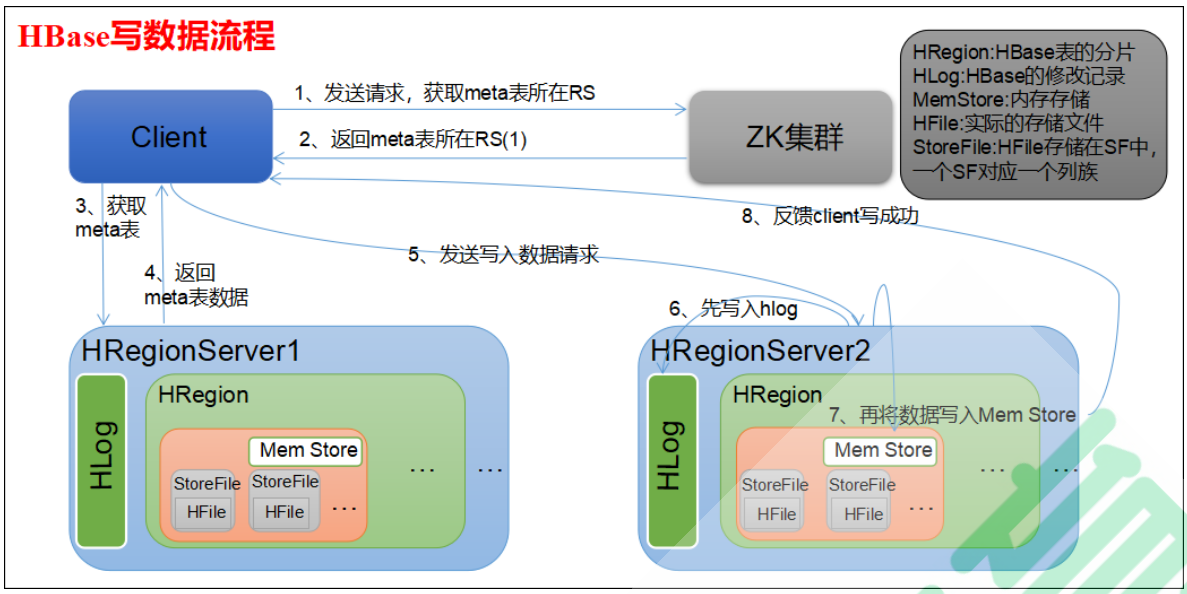
```
1、先从memstore找数据，如果没找到
2、从blockcache找数据，如果也没找到（布隆过滤器）
3、从HFile当中,HFile也是一种精妙设计的结果，扫描起来也不会特别的慢
```

```
HBase的单个key的扫描速度和  table 的 数据规模没有关系
```

为了能快速的判断要查询的数据在不在这个StoreFile中，应用了BloomFilter

```
BloomFilter，布隆过滤器：迅速判断一个元素是不是在一个庞大的集合内，但是他有一个弱点：它有一定的误判率
误判率：原本不存在与该集合的元素，布隆过滤器有可能会判断说它存在，但是，如果布隆过滤器，判断说某一个元素不存在该集合，那么该元素就一定不在该集合内。
```

1.4.2. 写请求过程



详细步骤：

```
1、Client先根据RowKey找到对应的Region所在的RegionServer
2、Client向RegionServer提交写请求
3、RegionServer找到目标Region
4、Region检查数据是否与Schema一致
5、如果客户端没有指定版本，则获取当前系统时间作为数据版本
6、将更新写入WAL Log
7、将更新写入Memstore
8、判断Memstore的是否需要flush为StoreFile文件。
```

```
写入数据的前提：数据写没写入都需要保证这张用户表是按照workey有序的
```

Hbase在做数据插入操作时，首先要找到RowKey所对应的的Region，怎么找到的？其实这个简单，因为.META.表存储了每张表每个Region的起始RowKey了。

建议：在做海量数据的插入操作，避免出现递增rowkey的put操作
如果put操作的所有RowKey都是递增的，那么试想，当插入一部分数据的时候刚好进行分裂，那么之后的所有数据都开始往分裂后的第二个Region插入，就造成了数据热点现象。

细节描述：HBase使用MemStore和StoreFile存储对表的更新。

1.4.3. Region的Split和Compact

数据在更新时首先写入HLog(WAL Log)，再写入内存(MemStore)中，
MemStore（CocurrentSkipListMap，优点就是 增删改查key-value效率都很高）中的数据是排序的，当MemStore累计到一定阈值(默认是128M，局部控制)时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时，系统会在ZooKeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了。当系统出现意外时，可能导致内存(MemStore)中的数据丢失，此时使用HLog(WAL Log)来恢复 checkpoint 之后的数据。

Memstore执行刷盘操作的的触发条件：

1、全局内存控制：当所有memstore占整个heap的最大比例的时候，会触发刷盘的操作。这个参数是hbase.regionserver.global.memstore.upperLimit，默认为整个heap内存的40%。这个全局的参数是控制内存整体的使用情况，但这并不意味着全局内存触发的刷盘操作会将所有的MemStore都进行输盘，而是通过另外一个参数hbase.regionserver.global.memstore.lowerLimit来控制，默认是整个heap内存的35%。当flush到所有memstore占整个heap内存的比率为35%的时候，就停止刷盘。这么做主要是为了减少刷盘对业务带来的影响，实现平滑系统负载的目的。

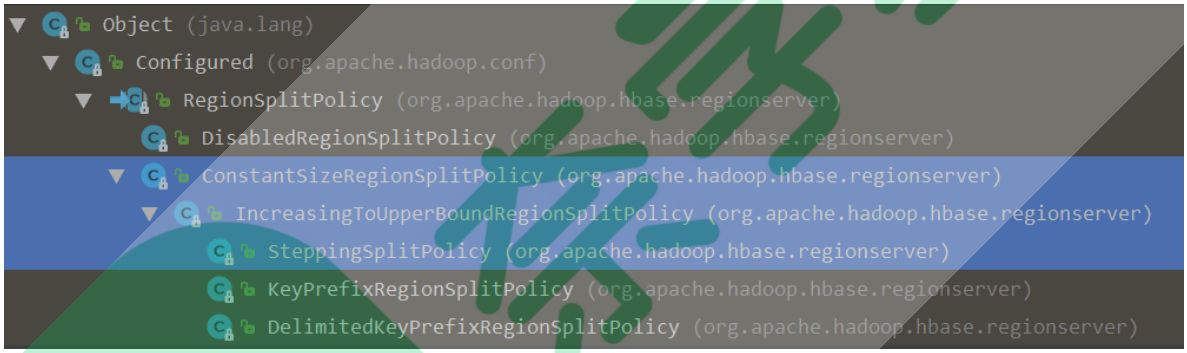
2、局部内存控制：当MemStore的大小达到hbase.hregion.memstore.flush.size大小的时候会触发刷盘，默认128M大小。

3、HLog的数量：前面说到HLog为了保证HBase数据的一致性，那么如果HLog太多的话，会导致故障恢复的时间太长，因此HBase会对HLog的最大个数做限制。当达到HLog的最大个数的时候，会强制刷盘。这个参数是hase.regionserver.max.logs，默认是32个。

4、手动操作：可以通过HBase Shell或者Java API手工触发flush的操作。

HBase 的三种默认的 Split 策略：

1、ConstantSizeRegionSplitPolicy　　常数数量
2、IncreasingToUpperBoundRegionSplitPolicy　　递增上限
3、SteppingSplitPolicy　　步增上线



StoreFile是只读的，一旦创建后就不可以再修改。因此HBase的更新/修改其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(minor_compact, major_compact)，将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对StoreFile进行split，等分为两个StoreFile。由于对表的更新是不断追加的，compact时，需要访问Store中全部的 StoreFile和MemStore，将他们按rowkey进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，合并的过程还是比较快。

Minor_Compact 和 Major_Compact 的区别：

（1）Minor操作只用来做部分文件的合并操作以及包括minVersion=0并且设置ttl的过期版本清理，不做任何删除数据、多版本数据的清理工作。
（2）Major操作是对Region下的HStore下的所有StoreFile执行合并操作，最终的结果是整理合并出一个文件。

Client写入 -> 存入MemStore，一直到MemStore满 -> Flush成一个StoreFile，直至增长到一定阈值 -> 触发Compact合并操作 -> 多个StoreFile合并成一个StoreFile，同时进行版本合并和数据删除 -> 当StoreFiles Compact后，逐步形成越来越大的StoreFile -> 单个StoreFile大小超过一定阈值后，触发Split操作，把当前Region Split成2个Region，Region会下线，新Split出的2个孩子Region会被HMaster分配到相应的HRegionServer 上，使得原先1个Region的压力得以分流到2个Region上。由此过程可知，**HBase只是增加数据，所有的更新和删除操作，都是在Compact阶段做的**，所以，用户写操作只需要进入到内存即可立即返回，从而保证I/O高性能。

写入数据的过程补充：

```
工作机制：每个HRegionServer中都会有一个HLog对象，HLog是一个实现Write Ahead Log的类，每次用户操作写入Memstore的同时，也会写一份数据到HLog文件，HLog文件定期会滚动出新，并删除旧的文件(已持久化到 StoreFile中的数据)。当HRegionServer意外终止后，HMaster会通过ZooKeeper感知，HMaster首先处理遗留的HLog文件，将不同Region的log数据拆分，分别放到相应Region目录下，然后再将失效的Region（带有刚刚拆分的log）重新分配，领取到这些Region的 HRegionServer在load Region的过程中，会发现有历史HLog需要处理，因此会Replay HLog中的数据到MemStore中，然后flush到StoreFiles，完成数据恢复。
```

1.5. RegionServer工作机制

1、Region分配

任何时刻，一个Region只能分配给一个RegionServer。master记录了当前有哪些可用的RegionServer。以及当前哪些Region分配给了哪些RegionServer，哪些Region还没有分配。当需要分配的新的Region，并且有一个RegionServer上有可用空间时，Master就给这个RegionServer发送一个装载请求，把Region分配给这个RegionServer。RegionServer得到请求后，就开始对此Region提供服务。

```
该regoin的compact和split
该region的IO读写
```

2、RegionServer上线

Master使用zookeeper来跟踪RegionServer状态。当某个RegionServer启动时，会首先在ZooKeeper上的server目录下建立代表自己的znode。由于Master订阅了server目录上的变更消息，当server目录下的文件出现新增或删除操作时，Master可以得到来自ZooKeeper的实时通知。因此一旦RegionServer上线，Master能马上得到消息。

3、RegionServer下线

当RegionServer下线时，它和zookeeper的会话断开，ZooKeeper而自动释放代表这台server的文件上的独占锁。Master就可以确定：

```
1、RegionServer和ZooKeeper之间的网络断开了。
2、RegionServer挂了。
```

无论哪种情况，RegionServer都无法继续为它的Region提供服务了，此时Master会删除server目录下代表这台RegionServer的znode数据，并将这台RegionServer的Region分配给其它还活着的同志。

1.6. Master工作机制

Master上线

Master启动进行以下步骤:

1、从ZooKeeper上获取唯一一个代表Active Master的锁，用来阻止其它Master成为Master。
使用zookeeper实现了分布式独占锁
2、扫描ZooKeeper上的server父节点，获得当前可用的RegionServer列表。
rs节点下的regionserver列表
3、和每个RegionServer通信，获得当前已分配的Region和RegionServer的对应关系。
每个表有多少个regoin，那些regionserver保管了那些region，..
4、扫描.META. Region的集合，计算得到当前还未分配的Region，将他们放入待分配Region列表。
有一些regoin是无人认领的。

Master下线

由于Master只维护表和Region的元数据，而不参与表数据IO的过程，Master下线仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的schema，无法进行Region的负载均衡，无法处理Region上下线，无法进行Region的合并，唯一例外的是Region的split可以正常进行，因为只有RegionServer参与)，表的数据读写还可以正常进行。因此Master下线短时间内对整个hbase集群没有影响。

从上线过程可以看到，Master保存的信息全是可以冗余信息（都可以从系统其它地方收集到或者计算出来）

因此，一般HBase集群中总是有一个Master在提供服务，还有一个以上的Master在等待时机抢占它的位置。